PYTHON

FILES AND EXCEPTIONS

NOR HAMIZAH MISWAN

READING FROM A FILE

- An incredible amount of data is available in text files.
- Text files can contain weather data, traffic data, socioeconomic data, literary works, and more.
- Reading from a file is particularly useful in data analysis applications, but it's also applicable to any situation in which you want to analyze or modify information stored in a file.
- When you want to work with the information in a text file, the first step is to read the file into memory.

READING FROM A FILE

- To do any work with a file, even just printing its contents, you first need to open the file to access it.
- The open() function needs one argument: the name of the file you want to open.

Try:

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
    print(contents)
```

- The open() function returns an object representing the file. Python stores this object in file_object, which we'll work with later in the program.
- The keyword with closes the file once access to it is no longer needed.

READING FROM A FILE

- If we want to looks file in other directory, set the file_path. How?

  file_path = 'C:/Users/Hamizah/Downloads/pi_digits.txt'

- Then use: with open(file_path) as file_object:

```
Try:
with open(file_path) as file_object:
    contents = file_object.read()
    print(contents)
```

- The open() function returns an object representing the file. Python stores this object in file_object, which we'll work with later in the program.

READING FROM A FILE

- Or directly paste the directory in the open () function:

Try:

```
with open('C:/Users/Hamizah/Downloads/pi_digits.txt') as file_object:
    contents = file_object.read()
    print(contents)
```

READING FROM A FILE

- Sometime we want to examine each line of the file. Especially when looking for certain information in the file, or might want to modify the text in the file in some way.
- You can use a for loop on the file object to examine each line from a file one at a time:

```
Try:
filename = 'pi_digits.txt'

with open(filename) as file_object:
    for line in file_object:  #use for loop
        print(line)
 #or use rstrip() function eliminate extra
blank lines
print(line.rstrip())
```

WRITING TO A FILE

- One of the simplest ways to save data is to write it to a file.
- When you write text to a file, the output will still be available after you close the terminal containing your program's output.
- Some ways to write a file:
  - Writing to an empty file
  - Writing multiple lines
  - Appending to a file

WRITING TO A FILE

- To write text to a file, you need to call open() with a second argument telling Python that you want to write to the file.
- The call to open() in this example has two arguments:
  - The first argument is still the name of the file we want to open.
  - The second argument, 'w', tells Python that we want to open the file in write mode.
- The open() function automatically creates the file you're writing to if it doesn't already exist.
- However, be careful opening a file in write mode ('w') because if the file does exist, Python will erase the file before returning the file object.

WRITING TO A FILE

Try:
filename = 'programming.txt'

```
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

- It will save the object as the filename and store it in the same directory.

WRITING TO A FILE

- The write() function doesn't add any newlines to the text you write.
- To write multiple lines, how?

Try:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.\n")
    file_object.write("I love creating new games.\n")
```

WRITING TO A FILE

- If you want to add content to a file instead of writing over existing content, you can open the file in append mode. How?

Try:

```
filename = 'programming.txt'

with open(filename, 'a') as file_object:
    file_object.write("I also love finding meaning in large datasets.\n")
    file_object.write("I love creating apps that can run in a browser.\n")
```

- When you open a file in append mode, Python doesn't erase the file before returning the file object. Any lines you write to the file will be added at the end of the file.

EXCEPTIONS

- Special objects Python creates to manage errors that arise while a program is running.
- Whenever an error occurs that makes Python unsure what to do next, it creates an exception object.
- Exceptions are handled with try-except blocks.
- A try-except block asks Python to do something, but it also tells Python what to do if an exception is raised.
- When you use try-except blocks, your programs will continue running even if things start to go wrong.

EXCEPTIONS

- Common error that causes Python to raise an exception:
  - ZeroDivisionError
  - FileNotFoundError
- Hence, we may write a try-except or try-except-else to handle these errors.

EXCEPTIONS

- Handling the ZeroDivisionError Exception:
  - You may try this:

```
print(5/0)
```

  - The above solution can be solved using try-except block:

```
try:
    print(5/0) #the code
except ZeroDivisionError:
    print("You can't divide by zero!") #print the error
```

EXCEPTIONS

- Handling the ZeroDivisionError Exception:
  - You may try this:

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break
    answer = int(first_number) / int(second_number)
    print(answer)
```

  - The above solution can be solved using try-except-else block:

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    try:
        answer = int(first_number) / int(second_number) #the
code
    except ZeroDivisionError:
        print("You can't divide by 0!") #print the error
    else:
        print(answer)
```

EXCEPTIONS

- Handling the FileNotFoundError Exception:

  ➢You may try this:

```
filename = 'alice.txt'

with open(filename) as f_obj:
    contents = f_obj.read()
```

➢The above solution can be solved using try-except block:

```
filename = 'alice.txt'

try:
    with open(filename) as f_obj: #the code for opening file
        contents = f_obj.read()
except FileNotFoundError:
    msg = "Sorry, the file " + filename + " does not exist."
    print(msg) #print the error
```

EXCEPTIONS

- In the previous example, we informed our users that one of the files was unavailable.
- But you don't need to report every exception you catch.
- Sometimes you'll want the program to fail silently when an exception occurs and continue on as if nothing happened. How?
  - Python has a pass statement that tells it to do nothing in a block

EXCEPTIONS

Try to handle error silently:

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    try:
        answer = int(first_number) / int(second_number)
    except ZeroDivisionError:
        pass
    else:
        print(answer)
```

STORING DATA

- When users close a program, you'll almost always want to save the information they entered.
- A simple way to do this involves storing your data using the json (JavaScript Object Notation) module:
  - The json module allows you to dump simple Python data structures into a file and load the data from that file the next time the program runs.
  - The json module can be used to share data between different Python programs.
  - The JSON data format is not specific to Python, so you can share data you store in the JSON format with people who work in many other programming

STORING DATA

- Let's write a short program that stores a set of numbers and another program that reads these numbers back into memory.
- The first program will use json.dump() to store the set of numbers.
  - The json.dump() function takes two arguments: a piece of data to store and a file object it can use to store the data.
  - Example: store a list of numbers

```
import json

numbers = [2, 3, 5, 7, 11, 13]
filename = 'numbers.json'
with open(filename, 'w') as f_obj:
    json.dump(numbers, f_obj)
```

- open the file in write mode allows json to write the data to the file.
- At w we use the json.dump() function to store the list numbers in the file numbers.json.

STORING DATA

- Now we'll write a program that uses json.load() to read the list back into memory.

```
import json

filename = 'numbers.json'
with open(filename) as f_obj:
    numbers = json.load(f_obj)

print(numbers)
```

  - The json.load() function to load the information stored in numbers.json, and we store it in the variable numbers.

PYTHON

TESTING YOUR CODES

NOR HAMIZAH MISWAN

TESTING A FUNCTION

- The module unittest from the Python standard library provides tools for testing your code.
- A unit test verifies that one specific aspect of a function's behavior is correct.
- A test case is a collection of unit tests that together prove that a function behaves as it's supposed to, within the full range of situations you expect it to hand.
- assert methods test whether a condition you believe is true at a specific point in your code is indeed true.

TESTING A FUNCTION

- Let's create a module named "name_function" and try to read from the module.

Module: name_function.py

```
def get_formatted_name(first, last):
    """Generate a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()
```

Read form the module:

```
from name_function import get_formatted_name
print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break
    formatted_name = get_formatted_name(first, last)
    print("\tNeatly formatted name: " +
formatted_name + '.')
```

TESTING A FUNCTION

- Then, try test the previous function using Passing Test.

```
import unittest
from name_function import get_formatted_name

class NamesTestCase(unittest.TestCase):
    """Tests for 'name_function.py'."""

    def test_first_last_name(self):
        """Do names like 'Janis Joplin' work?"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Joplin')

unittest.main()
```

TESTING A CLASS

- Python provides a number of assert methods in the unittest.TestCase class.
- As mentioned earlier, assert methods test whether a condition you believe is true at a specific point in your code is indeed true.
- If the condition is true as expected, your assumption about how that part of your program behaves is confirmed; you can be confident that no errors exist.
- There are six commonly used assert methods from unitest module.

TESTING A CLASS

Method	Use
`assertEqual(a, b)`	Verify that `a == b`
`assertNotEqual(a, b)`	Verify that `a != b`
`assertTrue(x)`	Verify that `x` is `True`
`assertFalse(x)`	Verify that `x` is `False`
`assertIn(item, list)`	Verify that *item* is in *list*
`assertNotIn(item, list)`	Verify that *item* is not in *list*

- You can use these methods only in a class that inherits from unittest .TestCase

NUMPY

ARRAY AND VECTORIZED COMPUTATION

NOR HAMIZAH MISWAN

NUMPY

- NumPy, short for Numerical Python, has long been a cornerstone of numerical computing in Python.
- It provides the data structures, algorithms, and library glue needed for most scientific applications involving numerical data in Python.
- NumPy contains, among other things:
  - A fast and efficient multidimensional array object ndarray
  - Functions for performing element-wise computations with arrays or mathematical operations between arrays
  - Tools for reading and writing array-based datasets to disk
  - Linear algebra operations, Fourier transform, and random number generation

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- One of the key features of NumPy is its N-dimensional array object, or ndarray,
  - a fast, flexible container for large datasets in Python.
- Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.
- To import numpy: import numpy as np
- Random number generation: np.random.randn(2, 3) #indicate 2 rows & 3 columns
- Creating ndarray: use the array function. How?

  np.array(list)

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- Other function for creating new array,
  - ➢ zeros and ones create arrays of 0s or 1s.

    Try: np.zeros(10) and np.zeros((3, 6))
  - ➢ arange is an array-valued version of the built-in Python range function.

    Try: np.arange(15)
- The data type or dtype is a special object containing the information (or metadata, data about data).

Try: arr1.dtype and arr2.dtype

Note: refer Table 4-2 for NumPy data types

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- Any arithmetic operations between equal-size arrays applies the operation element-wise. Try:

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
arr * arr
arr – arr
1 / arr
```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For indexing and slicing of NumPy array,
  - One-dimensional arrays are simple; on the surface they act similarly to Python lists.

  Try:

  ```
  arr = np.arange(10)
  arr[5:8] #index 5 to7
  arr[5:8] = 12 # replace index 5 to 7 with 12
  ```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For indexing and slicing of NumPy array,
  - Two dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays.
  - Thus, individual elements can be accessed recursively, by a comma-separated list of indices to select individual elements.

Try:

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d[2]  #index two
arr2d[0][2] #index at zero, with element of index two
arr2d[0, 2] #equivalent to above
```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For indexing and slicing of NumPy array,
  - In multidimensional arrays, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions.

  Try:

  ```
  arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
  arr3d[0] #2x3 array
  arr3d[1, 0] #index at one, whole element at index zero
  arr3d[1, 0,1]
  ```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- Ndarray can be sliced with familiar syntax [:].
  - Like one-dimensional objects such as Python lists, ndarrays can be sliced:

  Try: arr[1:6]
  - Slicing two-dimensional array is a bit different:

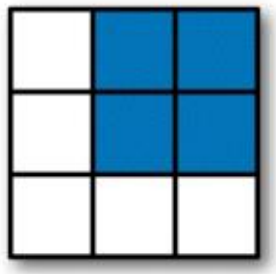
  Try: arr2d[:2]

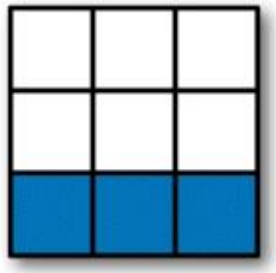
  arr2d[1, :2]
  - Multiple slices is just like you can pass multiple indexes:

  Try: arr2d[:2, 1:]

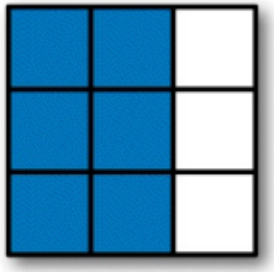
ndarray: MULTIDIMENSIONAL ARRAY OBJECT

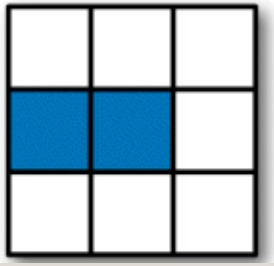
- For two-dimensional array slicing:





Expression	Shape
arr[:2, 1:]	(2, 2)
arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1, 3)





Expression	Shape
arr[:, :2]	(3, 2)
arr[1, :2]	(2,)
arr[1:2, :2]	(1, 2)

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For Boolean indexing,
  - Let's have some data in an array and an array of names with duplicates:

  Try:

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
data = np.random.randn(7, 4)
```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For Boolean indexing,
  - Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob‘:

Try:

```
names == 'Bob'
data[names == 'Bob']
data[names == 'Bob', 2:]
data[names == 'Bob', 3]
names != 'Bob‘
data[~(names == 'Bob')]
```

Note: The boolean array must be of the same length as the array axis it’s indexing

```
mask = (names == 'Bob') | (names == 'Will')
data[mask]
```

Note: To combine multiple Boolean conditions, use arithmetic: & (for and) and | (for or)

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For transposing arrays,
  - Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything.
  - Arrays have the transpose method and also the special T attribute

Try:

```
arr = np.arange(15).reshape((3, 5))
arr
Arr.T
np.dot(arr.T, arr)
```

UNIVERSAL FUNCTION: FAST ELEMENT-WISE ARRAY FUNCTION

- A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays.
- Many ufuncs are simple element-wise transformations, like sqrt or exp.
  - These are referred to as unary ufuncs.

  Try: arr = np.arange(10)
  np.sqrt(arr)
  np.exp(arr)
- Two arrays (binary ufuncs) return a single array as the result.
  - Example: add or maximum

  Try: x = np.random.randn(8),
  y = np.random.randn(8)
  np.maximum(x, y)

Note: Other unary and binary ufuncs can be obtained in Table 4-3 and Table 4-4

ARRAY-ORIENTED PROGRAMMING

- Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops.
- In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations.

Try:

```
points = np.arange(-5, 5, 0.01) # 1000 equally spaced points
xs, ys = np.meshgrid(points, points)
z = np.sqrt(xs ** 2 + ys ** 2)
```

ARRAY-ORIENTED PROGRAMMING

- A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class.

Try:

```
arr = np.random.randn(5, 4)
arr.mean() #equivalent to np.mean(arr)
arr.sum()
```

- Functions like mean and sum take an optional axis argument that computes the statistic over the given axis, resulting in an array with one fewer dimension:

Try:

```
arr.mean(axis=1) #axis=1 compute mean across the columns
arr.sum(axis=0) #axis=0 compute sum down the rows
```

Note: Other basic array statistical methods can be obtained in Table 4-5.

ARRAY-ORIENTED PROGRAMMING

- NumPy arrays can be sorted in-place with the sort method.

Try:

```
arr = np.random.randn(6)
arr.sort()
arr1 = np.random.randn(5, 3)
arr.sort(1)
```

- np.unique returns the sorted unique values in an array.

Try:

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
np.unique(names)
```

Note: Other array set operations can be obtained in Table 4-6.

FILE INPUT AND OUTPUT WITH ARRAYS

- NumPy is able to save and load data to and from disk either in text or binary format.
- np.save and np.load are the two workhorse functions for efficiently saving and loading array data on disk.
- Arrays are saved by default in an uncompressed raw binary format with file extension .npy. How?

```
arr = np.arange(10)
np.save('some_array', arr)
np.load('some_array.npy') #load the array
np.savez('array_archive.npz', a=arr, b=arr) #save multiple arrays in an uncompressed archive
arch = np.load('array_archive.npz')
arch[‘b’]
```

LINEAR ALGEBRA

- Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Try:

```
x = np.array([[1., 2., 3.], [4., 5., 6.]])
y = np.array([[6., 23.], [-1, 7], [8, 9]])
x.dot(y) #equivalent to np.dot(x, y)
```

- To consider a standard set of matrix decompositions and things like inverse and determinant, use numpy.linalg. Try:

```
from numpy.linalg import inv
X = np.random.randn(5, 5)
mat = X.T.dot(X)
inv(mat)
```

Note: Other commonly used numpy.linalg functions can be obtained in Table 4-6.

PSEUDORANDOM NUMBER GENERATION

- The numpy.random module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions (such as normal, beta, chi-square, gamma, binomial, etc.).
- For example, you can get a 4 × 4 array of samples from the standard normal distribution using normal:

```
samples = np.random.normal(size=(4, 4))
```

Note: Other commonly used list of numpy.random functions can be obtained in Table 4-6.